Strutture di controllo del flusso del programma C

CALCOLATORI ELETTRONICI – FONDAMENTI DI PROGRAMMAZIONE

a.a. 2024/2025

Prof. Roberto Pirrone

Espressioni logiche

- Le strutture di selezione Si basano sulla valutazione di *espressioni booleane o logiche*
  - Espressioni il cui valore è *vero* o *falso*
- In C:
  - Falso → l'espressione vale 0
  - Vero → l'espressione vale 1 (o comunque diverso da 0)

Espressioni logiche

- Le espressioni booleane sono costruite a partire da due tipologie di operatori:

  - Operatori relazionali

    Stabiliscono la verità di un confronto tra due sotto-espressioni

  - Operatori logici

    Connettono tra loro più sotto-espressioni secondo la logica classica

Espressioni logiche

- Operatori relazionali

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

Espressioni logiche

- Operatori logici

TABLE 4.3 The && Operator (and)

operand1	operand2	operand1 && operand2
nonzero (true)	nonzero (true)	1 (true)
nonzero (true)	0 (false)	0 (false)
0 (false)	nonzero (true)	0 (false)
0 (false)	0 (false)	0 (false)

Espressioni logiche

- Operatori logici

TABLE 4.4 The || Operator (or)

operand1	operand2	operand1 || operand2
nonzero (true)	nonzero (true)	1 (true)
nonzero (true)	0 (false)	1 (true)
0 (false)	nonzero (true)	1 (true)
0 (false)	0 (false)	0 (false)

Espressioni logiche

- Operatori logici

TABLE 4.5 The ! Operator (not) *unario*

operand1	!operand1
nonzero (true)	0 (false)
0 (false)	1 (true)

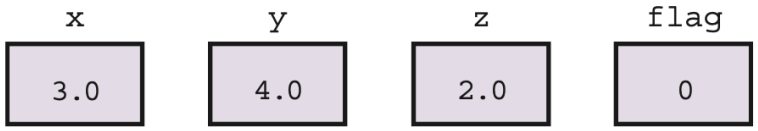
Espressioni logiche

- Precedenza degli operatori

È possibile inserire le parentesi anche per modificare la priorità dell'applicazione degli operatori nelle espressioni logiche.

Operator
function calls
`!` `+` `-` `&` (unary operators)
`*` `/` `%`
`+` `-`
`<` `<=` `>=` `>`
`==` `!=`
`&&`
`||`
`=`

Espressioni logiche




```
1.  !flag                              /* !0 is 1 (true)              */
2.  x + y / z  <=  3.5                 /* 5.0 <= 3.5 is 0 (false)     */
3.  !flag || (y + z  >=  x - z)        /* 1 || 1 is 1 (true)          */
4.  !(flag || (y + z  >=  x - z))      /* !(0 || 1) is 0 (false)      */
```

Espressioni logiche

- Valutazione *corto circuito*

- Le espressioni logiche vengono valutate da *sinistra a destra* (a meno di alterazioni dovute alle parentesi)

- *La valutazione si ferma non appena la verità dell'intera espressione è accertata*

Espressioni logiche

- Valutazione *corto circuito*

`a && b` è immediatamente falso se `a` è falso *senza valutare* `b`

`a || b` è immediatamente vero se `a` è vero *senza valutare* `b`

Espressioni logiche

- Appartenenza di una variabile ad un range

```
min <= x  &&  x <= max
```

- Confronto tra caratteri
  - `'9' >= '0'` vero
  - `'a' < 'e'` vero
  - `'B' <= 'A'` falso
  - `'Z' == 'z'` falso
  - `'a' <= 'A'` falso
  - `'a'<=ch && ch<='z'` vero se `ch` è una lettera minuscola (è un intervallo)

Espressioni logiche

- Il risultato di un'espressione logica può essere assegnato ad una variabile

```
int  age;             /* input - a person's age              */
char gender;          /* input - a person's gender           */
int  senior_citizen;  /* logical - indicates senior status   */

senior_citizen = 1;   /* Set senior status */
scanf("%d", &age);                        /* Read the person's age */
senior_citizen = (age >= 65);             /* Set senior status */
```

Espressioni logiche del genere sono valide:

```
!senior_citizen

senior_citizen  &&  gender == 'M'
```

Espressioni logiche

- Come negare un'espressione logica semplice del tipo:

exp = *op1* <relazionale> *op2* ?

- !*exp*

- *op1* <relazionale_complementare> *op2*
  - > diventa <=
  - < diventa >=
  - == diventa !=

Espressioni logiche

- Come negare un'espressione logica complessa (che usa `&&`, `||` e `!`)?
- Teorema di De Morgan
- `!(`*exp1* `&&` *exp2*`)` = *exp1_comp* `||` *exp2_comp*
- `!(`*exp1* `||` *exp2*`)` = *exp1_comp* `&&` *exp2_comp*

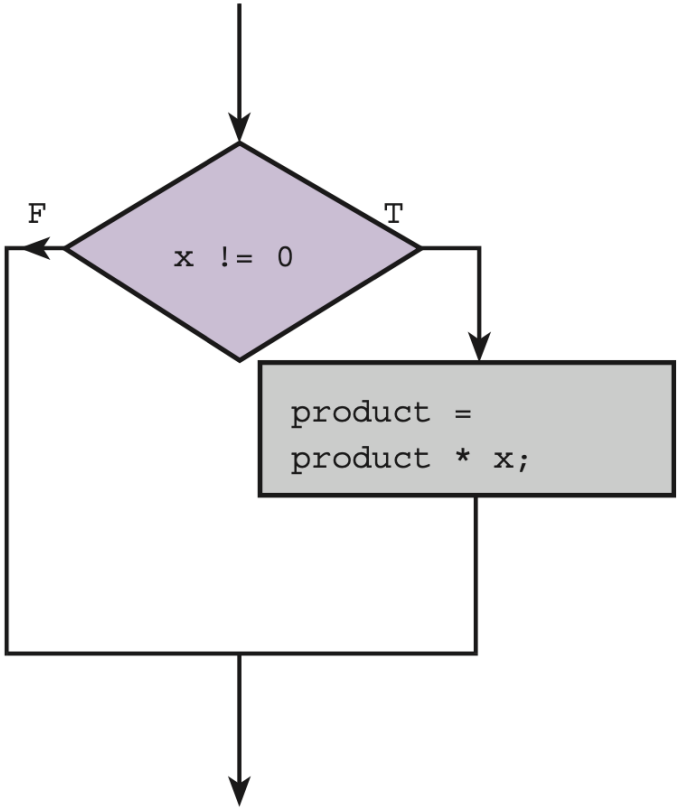
```
age > 25  &&  (status == 'S' || status == 'D')

age <= 25 || (status != 'S'  &&  status != 'D')
```

Strutture di selezione

- Selezione ad una alternativa

FORM: `if (`*condition*`)`
statement$_T$`;`

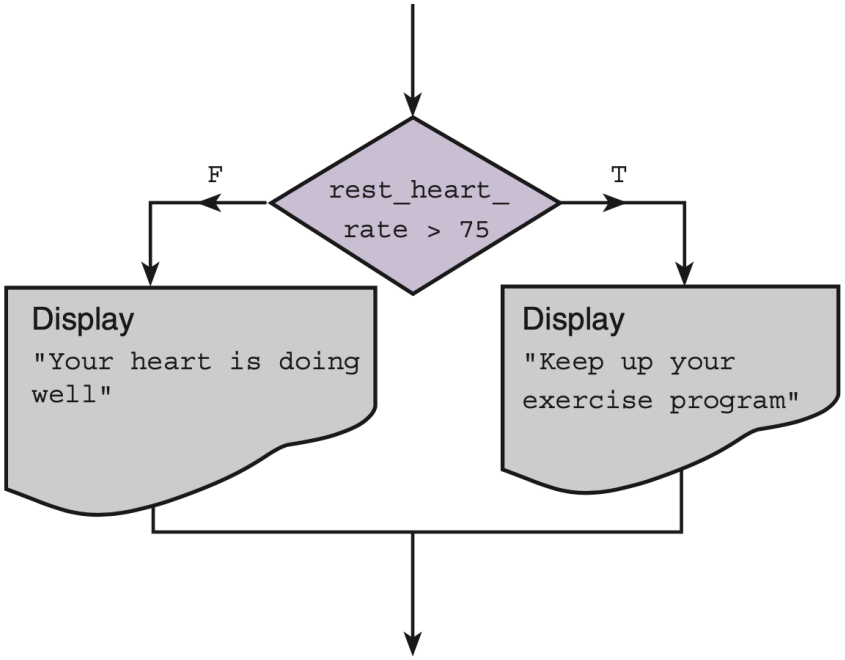
EXAMPLE:
```
if (x > 0.0)
    pos_prod = pos_prod * x;
```



Strutture di selezione

- Selezione a due alternative

FORM:
```
if (condition)
    statement_T;
else
    statement_F;
```
EXAMPLE:
```
if (x >= 0.0)
    printf("positive\n");
else
    printf("negative\n");
```



Strutture di selezione

- Selezione a più alternative

SYNTAX: `if (`*condition*$_1$`)`
statement$_1$
`else if (`*condition*$_2$`)`

```
if (x > 0)
        num_pos = num_pos + 1;
else

        if (x < 0)
                num_neg = num_neg + 1;
        else /* x equals 0 */
                num_zero = num_zero + 1;
```

E>

```
                num_neg = num_neg + 1;
        else /* x equals 0 */
                num_zero = num_zero + 1;
```

Università degli Studi di Palermo

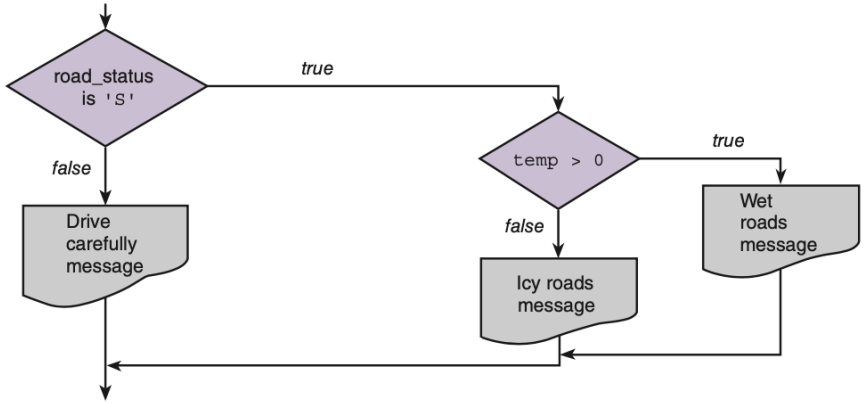
dipartimento di ingegneria unipa

CHILAB

Strutture di selezione

- Selezione tra più alternative

```
if (road_status == 'S')
      if (temp > 0) {
            printf("Wet roads ahead\n");
            printf("Stopping time doubled\n");
      } else {
            printf("Icy roads ahead\n");
            printf("Stopping time quadrupled\n");
      }
else
      printf("Drive carefully!\n");
```



Strutture di selezione: `switch`

SYNTAX: `switch` (*controlling expression*) {

label set$_1$

statements$_1$

`break;`

label set$_2$

statements$_2$

`break;`

. . .

label set$_n$

statements$_n$

`break;`

`default:`

statements$_d$

}

EXAMPLE:

```
/* Determine life expectancy in hours of an
   incandescent light bulb */
switch (watts) {
case 25:
      life = 2500;
      break;

case 40:
case 60:
      life = 1000;
      break;

case 75:
case 100:
      life = 750;
      break;

default:
      life = 0;
}
```

Solo di tipo `int` o `char`

Interrompe l'esecuzione

Caso standard, quando l'espressione di controllo non è uguale a nessuna etichetta

Università degli Studi di Palermo

dipartimento di ingegneria unipa

CHILAB

Iterazioni

- Una iterazione o ciclo è controllato tramite una espressione logica e consta sempre di tre componenti:
  - Una istruzione di inizializzazione della condizione di controllo
  - La vera e propria condizione che itera il ciclo se è vera
  - Una istruzione di *aggiornamento* della verità della condizione per terminare il ciclo

Iterazioni

- Sintassi delle strutture di controllo iterative

`for` (*espr. di inizializzazione* **;**
espr. di controllo **;**
espr. di aggiornamento)
istruzioni da iterare **;**

Iterazioni

- Sintassi delle strutture di controllo iterative

espr. di inizializzazione ;

`while(`*espr. di controllo*`)`

`{`

istruzioni da iterare ;

espr. di aggiornamento ;

`}`

Iterazioni

- Sintassi delle strutture di controllo iterative

```
do
{
        espr. di inizializzazione;  /* è parte delle
                                       istruzioni da iterare */
        istruzioni da iterare;
        espr. di aggiornamento;
}
while(espr. di controllo)
```

Iterazioni

- I cicli vengono usati in particolari condizioni tipiche:
  - Cicli a conteggio → conosco a priori il numero di iterazioni
  - Cicli condizionali generici → l'iterazione continua finché una certa condizione non si avvera
  - Cicli sentinella → iterazioni di input di dati che continuano finché non si inserisce un *valore speciale* (valore *sentinella*)

Iterazioni

- I cicli vengono usati in particolari condizioni tipiche:
  - Cicli controllati dalla fine di un file → iterazioni di input di valori da file che continuano finché non si raggiunge la fine (in C la costante `EOF`)
  - Cicli di validazione degli ingressi → l'iterazione continua finché l'input dell'utente non ricade in un certo range richiesto

Iterazioni a conteggio

```
/* Compute the payroll for a company */

#include <stdio.h>

int
main(void)
{
    double total_pay;     /* company payroll        */
    int    count_emp;     /* current employee       */
    int    number_emp;    /* number of employees    */
    double hours;         /* hours worked           */
    double rate;          /* hourly rate            */
    double pay;           /* pay for this period    */
```

Iterazioni a conteggio

```
/* Get number of employees. */
printf("Enter number of employees> ");
scanf("%d", &number_emp);

/* Compute each employee's pay and add it to the payroll. */
total_pay = 0.0;
count_emp = 0;
while (count_emp < number_emp) {
    printf("Hours> ");
    scanf("%lf", &hours);
    printf("Rate > $");
    scanf("%lf", &rate);
    pay = hours * rate;
    printf("Pay is $%6.2f\n\n", pay);
    total_pay = total_pay + pay;                /* Add next pay. */
    count_emp = count_emp + 1;
}
printf("All employees processed\n");
printf("Total payroll is $%8.2f\n", total_pay);

return (0);
}
```

Iterazioni a conteggio

- Operatori di assegnamento composti

- Servono ad abbreviare istruzioni del tipo: *var* = *var* <op> *exp*

```
+=
-=
*=
/=
%=
```

```
x = x * 4.56 → x *= 4.56
```

Iterazioni a conteggio

```
/* Process payroll for all employees */
total_pay = 0.0;
for (count_emp = 0;                    /* initialization                */
     count_emp < number_emp;           /* loop repetition condition     */
     count_emp += 1) {                 /* update                        */
    printf("Hours> ");
    scanf("%lf", &hours);
    printf("Rate > $");
    scanf("%lf", &rate);
    pay = hours * rate;
    printf("Pay is $%6.2f\n\n", pay);
    total_pay = total_pay + pay;
}
printf("All employees processed\n");
printf("Total payroll is $%8.2f\n", total_pay);
```

Iterazioni a conteggio

- Operatori di auto-incremento e auto-decremento

- Semplificano ulteriormente gli incrementi/decrementi unitari delle variabili di conteggio

  ++

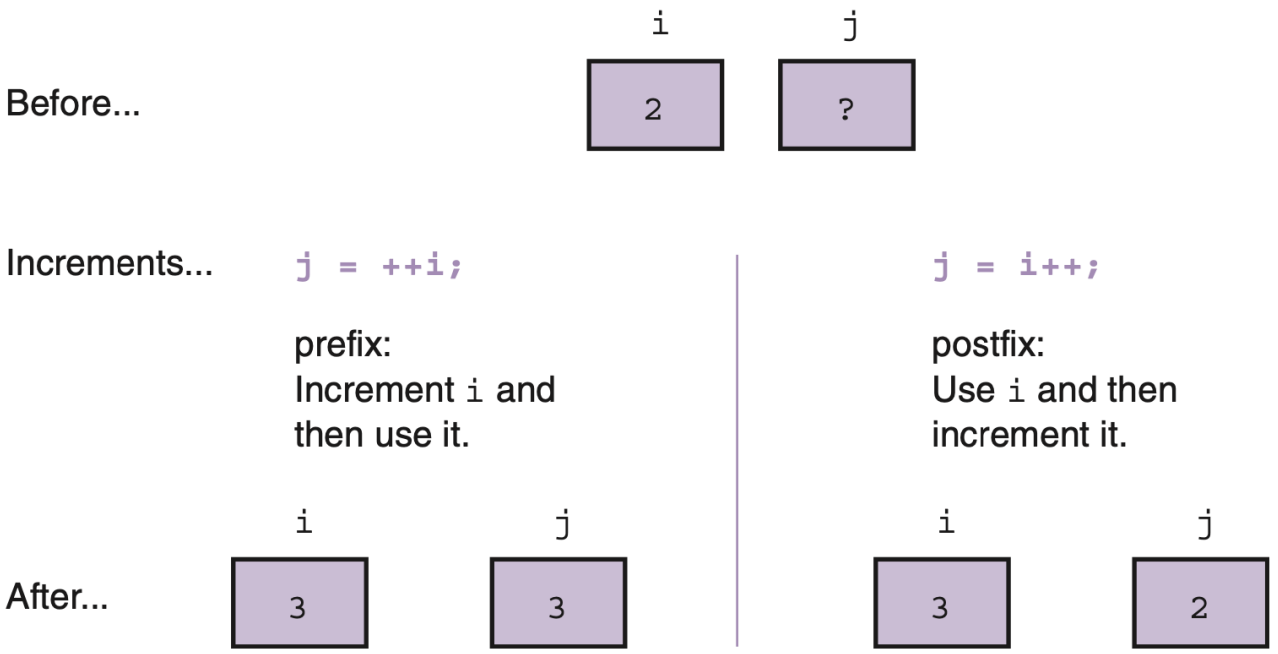
  −−

- Sono operatori unari e possono essere in posizione *prefissa* o *postfissa*

Iterazioni a conteggio

- ++*expr* prima incrementa *expr* di 1 e poi valuta il risultato
- −−*expr* prima decrementa *expr* di 1 e poi valuta il risultato
- *expr*++ prima valuta *expr* e poi la incrementa di 1
- *expr*−− prima valuta *expr* e poi la decrementa di 1

Iterazioni a conteggio



Iterazioni sentinella

```
/* Compute the sum of a list of exam scores. */

#include <stdio.h>

#define SENTINEL -99

int
main(void)
{
        int sum = 0,    /* output - sum of scores input so far              */
            score;      /* input - current score                            */

        /* Accumulate sum of all scores.                                    */
        printf("Enter first score (or %d to quit)> ", SENTINEL);
        scanf("%d", &score);        /* Get first score.                     */
        while (score != SENTINEL) {
            sum += score;
            printf("Enter next score (%d to quit)> ", SENTINEL);
            scanf("%d", &score);  /* Get next score.                   */
        }
        printf("\nSum of exam scores is %d\n", sum);

        return (0);
}
```

Si usa un valore convenzionale che non ricade nel range degli ingressi

È necessario acquisire un primo valore fuori dal ciclo

Iterazioni sentinella

```
/* Accumulate sum of all scores.                                    */
printf("Enter first score (or %d to quit)> ", SENTINEL);
for  (scanf("%d", &score);
      score != SENTINEL;
      scanf("%d", &score)) {
    sum += score;
    printf("Enter next score (%d to quit)> ", SENTINEL);
}
```

Iterazioni controllate dalla fine del file

```
#include <stdio.h>

int
main(void)
{
      int sum = 0,      /* sum of scores input so far */
          score,        /* current score  */
          input_status; /* status value returned by scanf */

      printf("Scores\n");

      input_status = scanf("%d", &score);
      while (input_status != EOF) {
            printf("%5d\n", score);
            sum += score;
            input_status = scanf("%d", &score);
      }

      printf("\nSum of exam scores is %d\n", sum);

      return (0);
}
```

`EOF` è un valore sentinella

Iterazioni di validazione degli ingressi

```
/*
 * Returns the first integer between n_min and n_max entered as data.
 * Pre : n_min <= n_max
 * Post: Result is in the range n_min through n_max.
 */
int
get_int (int n_min, int n_max)
{
        int   in_val,                /* input - number entered by user    */
              status;                /* status value returned by scanf    */
        char  skip_ch;               /* character to skip                 */
        int   error;                 /* error flag for bad input          */
```

Iterazioni di validazione degli ingressi

```
/* Get data from user until in_val is in the range.                */
do {
     /* No errors detected yet. */
     error = 0;
     /* Get a number from the user. */
     printf("Enter an integer in the range from %d ", n_min);
     printf("to %d inclusive> ", n_max);
     status = scanf("%d", &in_val);

     /* Validate the number. */
     if (status != 1) { /* in_val didn't get a number */
         error = 1;
         scanf("%c", &skip_ch);
         printf("Invalid character >>%c>>. ", skip_ch);
         printf("Skipping rest of line.\n");
     } else if (in_val < n_min || in_val > n_max) {
         error = 1;
         printf("Number %d is not in range.\n", in_val);
     }

     /* Skip rest of data line. */
     do
          scanf("%c", &skip_ch);
     while (skip_ch != '\n');
} while (error);

return (in_val);
}
```